A Formal Language for Cryptographic Protocol Requirements *

Paul Syverson and Catherine Meadows
Center for High Assurance Computing Systems
Naval Research Laboratory
Washington, DC 20375
USA

Abstract

In this paper we present a formal language for specifying and reasoning about cryptographic protocol requirements. We give sets of requirements for key distribution protocols and for key agreement protocols in that language. We look at a key agreement protocol due to Aziz and Diffie that might meet those requirements and show how to specify it in the language of the NRL Protocol Analyzer. We also show how to map our formal requirements to the language of the NRL Protocol Analyzer and use the Analyzer to show that the protocol meets those requirements. In other words, we use the Analyzer to assess the validity of the formulae that make up the requirements in models of the protocol. Our analysis reveals an implicit assumption about implementations of the protocol and reveals subtleties in the kinds of requirements one might specify for similar protocols.

Introduction

The past few years have seen a proliferation of formal techniques for the specification and analysis of cryptographic protocols. That these techniques can be useful has been shown by the fact that several (including BAN logic [5], the NRL Protocol Analyzer [12] [13], and the Stubblebine-Gligor model [16]) have been used to find flaws in open literature protocols that were previously believed to have been secure. Thus the use of formal methods for the analysis of cryptographic protocols has begun to attract attention as a promising way of guaranteeing their correctness.

Less attention, however, has been paid to the question of what exactly constitutes the correctness of a cryptographic protocol. Yet, we see that what constitutes correctness can vary widely with the application. In a key distribution protocol guarantee of secrecy and guarantee against replay attacks and impersonation are of the most importance. For a protocol used to guarantee the security of banking deposits, secrecy may or may not be important, although guarantee against replay attacks and impersonation definitely will be. Guarantee of timeliness may also be important, as well as the guarantee that messages are processed in the order that they are sent. (For example, a malicious intruder could cause somebody to overdraw his account by causing a deposit message and a withdrawal message to processed out of order.) For a protocol used to

*Earlier versions of portions of this paper have appeared in [17] and [18].

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **1996**	2. REPORT TYPE	3. DATES COVERED **00-00-1996 to 00-00-1996**

4. TITLE AND SUBTITLE **A Formal Language for Cryptographic Protocol Requirements**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Research Laboratory,Center for High Assurance Computer Systems,4555 Overlook Avenue, SW,Washington,DC,20375**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR’S ACRONYM(S)
	11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**		**30**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

distribute rights by proxy, it might be necessary not only to guarantee against impersonation but also to guarantee the entire pedigree of a message.

Protocols may also differ in the amount of trust that is placed in each individual. For example, Burrows, Abadi, and Needham, in their logic of authentication, make the assumption that the parties trying to authenticate each other are honest and will follow the rules of the protocol.[1] For other protocols, this may not necessarily be the case. In the Burns-Mitchell resource sharing protocol [4], it is assumed that the party attempting to obtain the resource may be trying to cheat the resource supplier into giving him a resource that he has not paid for at the same time he is trying to guarantee the the resource supplier is not cheating him. In a voting protocol, we make the assumption that individuals may try to find out other individuals' votes, that they may try to cast their votes more than once, and that they may be willing to divulge their votes to a small group of individuals if this will help them subvert the goals of the protocol.

Even when we restrict ourselves to the analysis of key distribution protocols, it is not always clear what constitutes the appropriate requirements. For example, in [6], Burrows, Abadi, and Needham describe the various orders of belief that a protocol can achieve, but make no recommendations. A protocol may achieve first order belief, in which A believes that K is a good key for communication with B, and vice versa, but neither has any belief about the beliefs of the other, or it may achieve second order belief, in which not only does each believe in the key, but each believes the other believes in the key, or it may achieve some yet higher order of belief. In [19] Syverson discusses the various orders of belief and where each would be appropriate.

Misunderstandings about requirements and assumptions has also contributed to much of the controversy about the various techniques. For example, in [15], Nessett points out an alleged flaw in the Burrows-Abadi-Needham logic by using it to prove that a protocol in which keys are distributed in an obviously unsafe way is secure. The response of Burrows, Abadi, and Needham [6] was that in their logic they make the assumption that principals do not divulge their keys; since in this protocol the principals do divulge their keys, it does not satisfy the original assumption. But one can also argue that the use of an unsafe key distribution method is not the same as knowingly divulging your key.

The degree to which requirements and assumptions can vary, and the controversy that can be caused by a lack of precise understanding of what the requirements are, suggests that we need to pay more attention to understanding and stating them in a precise way. Once we have a clear and precise statement of what the goals and assumptions of a protocol are, we can attempt to prove it satisfies these goals with a high degree of confidence that we know what we are about.

In this paper we attempt to make it easier to state and reason about requirements in a precise manner by providing a requirements specification language for the NRL Protocol Analyzer. The NRL Protocol Analyzer has the advantage that it is tied to no fixed set of assumptions about the kinds of requirement it is used to verify. The specifier of a protocol can use it to prove that an insecure state is not reachable, or that an insecure sequence of events cannot occur; it is up to the specifier to decide what these states and sequences are. However, until now the user of the Analyzer had to specify the undesired states and sequences in terms of the protocol specification itself. Thus the requirements had to be rewritten for each protocol specification, even when the aims of the protocols were identical. With the requirements specification language, it is possible to specify a set of requirements for a class of protocols, and then map them to a particular instantiation. It is also possible to reason about the requirements in isolation without concerning ourselves with particular

[1]Honesty assumptions are relaxed in the version of BAN presented in [2].

protocol instantiations.

An Example

In order to make clearer the abstract constructs we describe in this paper we set out a specific protocol as an example. We present the well known Kerberos protocol. [14] We set this out simply for illustrative purposes; thus we present a simplified version of the protocol as first given in [5].

The Kerberos Protocol

(1) *A sends to S*: A, B
(2) *S sends to A*: $\{T_S, L, K_{AB}, B, \{T_s, L, K_{ab}, A\}_{K_{BS}}\}_{K_{AS}}$
(3) *A sends to B*: $\{T_S, L, K_{AB}, A\}_{K_{BS}}, \{A, T_A\}_{K_{AB}}$
(4) *B sends to A*: $\{T_A + 1\}_{K_{AB}}$

Here A and B are two principals. By sending the first message, A requests of the authentication/key server S that a key K_{AB} be distributed for a secure communication session between A and B. Curly braces indicate encryption and subscripting with a key indicates the key used therein. Thus, the second message is entirely encrypted with K_{AS}, a key good for secure communication between A and the server. It contains a timestamp generated by S, T_S, a lifetime L for the session key, the session key itself, the name B, and another field that is encrypted with K_{BS} a key good for secure communication between B and S. The second message gives A the session key and lets A know that this was freshly sent by S and is for a session with B. The third message gives B the session key and lets him know that it was freshly sent by S. The second part of the third message shows B that A has the session key. The last message lets A know that B has recently seen the session key. This is an example of the type of protocol to which our first set of formal requirements is addressed.

The remainder of this paper is organized as follows. In section 1 we present an example of the type of requirements for which we intend to use our language. Specifically, we look at two party key distribution protocols involving a single key/authentication server. In section 2 we present the language itself and a corresponding model-theoretic semantics. We also present the model of computation on which our semantics is based, namely that underlying the NRL Protocol Analyzer. In section 3 we give a brief overview of the Analyzer. In section 4 we give requirements for two party key agreement protocols, such as in Diffie-Hellman key agreement [7]. We also look at a particular key agreement protocol due to Aziz and Diffie [3]. We specify this protocol in the language of the Analyzer and evaluate it with respect to the requirements we have given. Finally, we discuss our findings and present our conclusions.

1 Formal Requirements

One of the disadvantages of currently available logical languages for cryptographic protocol analysis is that for the most part each protocol has its own specification. Our approach goes some way towards a remedy by allowing a single set of requirements to specify a whole class of protocols. This has the advantage that a protocol analyst can largely identify the goals of any protocol in this class with that one specification, which seems to be a fairly intuitive way to view things. For instance one might want evaluate a protocol for two party session key distribution using ordinary public or shared key cryptography. While many of these protocols have special features and requirements, there are a number of requirements they all share—for example, that the distributed key be known only to the two principals and the server if there is one. We can express in our language general requirements for protocols for distributing session keys to two parties via a server. This specification should also satisfy protocols with no server, i.e., where one of the participants is the server. It should also work for interdomain communications. Although there are undoubtedly further requirements to be specified for servers from different domains to authenticate each other, that process should not affect the requirements for the two end parties in relation to whoever produces the key.

We begin by setting out formal requirements for key distribution protocols such as Kerberos. We will give a gloss of the notation that should be sufficient for an intuitive reading of the requirements. Precise syntax and semantics for our language will follow. After that we will consider requirements where the session key is the result of input from both recipients of the session key (as in Diffie-Hellman key exchange).

1.1 Two Party, One Server Key Distribution Requirements

What are the general security requirements for the type of authentication protocol given by our example above? That is, if A and B were to accept K_{ab} as a usable session key, what need hold to preclude security violations? We restrict outselves to the case in which there are two parties involved in obtaining keys, one who initiates the protocol, who we designate as the initiator, and the other, who we designate as the receiver. The server can be either a separate entity, or the initiator or receiver.

For this set of requirements, we assume that the server (given that he is distinct from the two principals) is honest. Individuals attempting to communicate may be either honest or dishonest. However, we only consider requirements for communication between two honest principals together with an honest server. This is because, under our assumptions, if any party is dishonest, he or she will share the key with the intruder, and so the fundamental requirement of key secrecy will not be satisfied.[2]

There are some obvious requirements on such a protocol. First of all, if a key is accepted by an honest principal for communication with another honest principal, it should not be learned by the intruder, either before or after the accept event, unless as a result of some key compromise that is outside the scope of the protocol. Secondly, replays of old keys should be avoided. Thus, if a key is accepted for communication by honest principal A with honest principal B, it should not have been accepted in the past, except possibly by B for communication with A. Thirdly, if a key has been accepted for communication between A and B, then it should have been generated by a server for use between A and B. Finally, we make the more subtle requirement that, if A or B accept a key for conversation with the other and with A as an initiator, then A did in fact initiate the conversation. Thus, A and B cannot be tricked into having a conversation that neither one of them initiated.

Events are expressed in our language using n-ary action symbols, often of arity four. The first argument of the action is reserved for the agent of that action. The next argument(s) are other parameters such as a key

[2] In other cases, for example in our analysis of some resource-sharing protocols, we develop requirements for the interaction of an honest principal with a possibly dishonest principal.

being distributed or another relevant principal. The last argument is always a round number local to the agent of the action. The events in the informal requirements that we have stated so far are: the initiator's request for a key for communication with the receiver, the server's sending a key for sender and receiver, the initiator's acceptance of a key for communication with the receiver, the receiver's acceptance of a key for communication with the initiator, the intruder's learning a key, and the compromise of a key.

They are:

- Initiator A requests to talk to receiver B:
  $\texttt{request}(user(A, honest), user(B, Y), (), M)$
- Server S sends a key K for communication between A and B:
  $\texttt{send}(S, user(A, X), user(B, Y), K, M)$
- Initiator A accepts a key for conversation with receiver B:
  $\texttt{init\_accept}(user(A, honest), user(B, Y), K, M)$
- Receiver B accepts a key for conversation with initiator A:
  $\texttt{rec\_accept}(user(B, honest), user(A, X), K, M)$
- Penetrator P learns a key:
  $\texttt{learn}(P, (), K, M)$
- Key is compromised:
  $\texttt{compromise}(environment, (), K, M)$

The requirements language uses standard logical notaion in the usual way. (For example, '$\rightarrow$' represents the standard logical conditional, and '$\wedge$' is the conjunction sign). The language also contains a temporal operator, $\diamondminus$, which can be read "at some point in the past". Our use of '$M?$' below is not as a true variable, hence it does not require uniform substitution of round numbers.

The requirements are:

Requirement 1 If a key has been accepted, it should not be learned by the intruder, except through a compromise event:

$$\begin{aligned}&\diamondminus(\texttt{init\_accept}(user(A, honest), user(B, honest), K, M1)\vee\\ &\qquad\texttt{rec\_accept}(user(B, honest), user(A, honest), K, M2)) \rightarrow\\ &\diamondminus(\texttt{learn}(pen, (), K, M?) \rightarrow \diamondminus\texttt{compromise}(environment, (), K, M?))\end{aligned}$$

Probably the main feature in Kerberos corresponding to this requirement is that the session key is sent only in encrypted form—and the assumption that that principals will not release the key themselves. We note this only for illustrative purposes. We do not claim to fully describe the features of Kerberos, nor do we analyze the protocol.

Requirement 2 If a key is accepted for communication between two parties, it should not have been accepted in the past, except by the other party. This becomes two requirements, one for the initiator and one for the receiver. (The receiver's requirement is a mirror image of the sender's.)

$$\begin{array}{l}\texttt{init\_accept}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), K, M1) \rightarrow \\ \quad \neg(\diamondminus \texttt{init\_accept}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?) \wedge \\ \quad (\diamondminus \texttt{rec\_accept}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?) \rightarrow (C = B \wedge D = A))\end{array}$$

$$\begin{array}{l}\texttt{rec\_accept}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), K, M2) \rightarrow \\ \quad \neg(\diamondminus \texttt{rec\_accept}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?) \wedge \\ \quad (\diamondminus \texttt{init\_accept}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?) \rightarrow (C = A \wedge D = B))\end{array}$$

Given the assumptions that principals will follow the protocol faithfully and that the server always generates new keys, probably the main feature of Kerberos relating to this requirement is the use of timestamps that are bound cryptographically to K_{ab} in one form or another.

Requirement 3 If a key is accepted for communication between two entities, then it must have been requested by the initiating entity and sent by a server for communication between those two entities. Again, this becomes two requirements, one for the initiator and one for the receiver.

$$\begin{array}{l}\texttt{init\_accept}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), K, M1) \rightarrow \\ \quad \diamondminus(\texttt{send}(S, (\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest})), K, M?) \wedge \\ \quad\quad \diamondminus \texttt{request}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), (), M1))\end{array}$$

$$\begin{array}{l}\texttt{rec\_accept}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), K, M2) \rightarrow \\ \quad \diamondminus(\texttt{send}(S, (\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest})), K, M?) \wedge \\ \quad\quad \diamondminus \texttt{request}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), (), M?))\end{array}$$

Again assuming that principals execute the protocol faithfully, the use of timestamps, appropriately bound to K_{ab}, forms the primary mechanisms of Kerberos that relate to this requirement.

In order to be secure a protocol must satisfy the conjunction of the requirements. They must all hold; although, it helps keep things clear if we list them separately. This will also facilitate application of the NRL Protocol Analyzer.

Note that according to the requirements it does not matter whether a protocol uses public or shared key cryptography. Nor do we specifically require which freshness mechanisms are used. The Kerberos Protocol set out above uses shared keys and timestamps, but the requirements apply equally to protocols using public keys and/or nonces or perhaps sequence numbers. These points should provide some indication of the generality with which requirements can be stated even when being formal. We will return to the discussion of requirements below, after we have precisely set out the language and its interpretation.

2 The Language

In this section we set out our formal requirements language. In general, our syntax is based on that of temporal logic (cf. [10] or [21]) and in particular was motivated by the language of [11] and [1]; however, the intended meaning of the syntax is somewhat different than in those works. We begin with the general linguistic constructs and then give the interpretation thereof in the model of computation.

2.1 Syntax

Our language contains a denumerable collection of constant singular terms, typically represented by letters from the beginning of the alphabet. We also have a denumerable collection of variable terms, typically represented by letters from the end of the alphabet. We also have, for each $n \geq 1$, n-ary function letters taking terms of either type as arguments and allowing us to build up functional terms in the usual recursive fashion. (We will always indicate whether a term is constant or variable if there is any potential for confusion.) We have a denumerable collection of n-ary action symbols for each arity $n \geq 1$. These will be written as words in typewriter script (e.g., `accept`). The first argument of an action symbol is reserved for a term representing the agent of the action in question.

An atomic formula consists of an n-ary action symbol, e.g., '`act`' followed by an n-tuple of terms. We have the usual logical connectives: $\neg$, $\wedge$, $\vee$, $\rightarrow$, and $\leftrightarrow$, and also one temporal operator: $\diamondminus$. Complex formulae are built up from atomic formulae in the usual recursive fashion. Since we have already seen examples of formulae, we proceed directly to their interpretation. (Note that this is only a formal language, not a logic; hence there are no axioms or inference rules.)

2.2 Interpretations

The key notion to understand is that of an action. For us actions are transitions from one state to another. We represent these semantically by ordered pairs of the form (s, s'), where 's' represents the state prior to the action and 's'' represents the state subsequent to the action. The precise way this works is given in the definition of an interpretation.

Definition 2.1 A *state space* is a non-empty set S, and each $s \in S$ is a *state*. We represent time digitally using the integers. A *trace* is a sequence σ of elements of S that is infinite in both directions, for example, $\ldots, s_{i-1}, s_i, s_{i+1}, \ldots$. We can thus equate a trace with a function from times to states. If s is the value of $\sigma(t)$, we will generally adopt the notational convenience of representing this by 's_t'. Let α and β be formulae. An *interpretation* is a function I from atomic formulae of the language to subsets of $S \times S$, i.e., $I(\alpha) \subseteq S \times S$ for any atomic formula α.

A *model* is an ordered 4-tuple, $\langle S, I, \sigma, t\rangle$ such that S is a state space, I is an interpretation, σ is a trace, and t is a time. The *satisfaction relation*, $\models$, is a relation between models and formulae. It is our way of specifying which formulae are true: given a formula α and a model $\langle S, I, \sigma, t\rangle$, '$\langle S, I, \sigma, t\rangle \models \alpha$' means that α is true at $\langle S, I, \sigma, t\rangle$. It is defined as the smallest relation between models and formulae satisfying the following:

$$
\begin{aligned}
\langle S, I, \sigma, t\rangle \models \alpha \quad &=_{df} \quad (s_t, s_{t+1}) \in I(\alpha) \\
\langle S, I, \sigma, t\rangle \models \neg\alpha \quad &=_{df} \quad \langle S, I, \sigma, t\rangle \not\models \alpha \\
\langle S, I, \sigma, t\rangle \models \alpha \wedge \beta \quad &=_{df} \quad \langle S, I, \sigma, t\rangle \models \alpha \text{ and } \langle S, I, \sigma, t\rangle \models \beta
\end{aligned}
$$

$$
\begin{aligned}
\langle S, I, \sigma, t\rangle \models \alpha \vee \beta &=_{df} \langle S, I, \sigma, t\rangle \models \alpha \text{ or } \langle S, I, \sigma, t\rangle \models \beta \\
\langle S, I, \sigma, t\rangle \models \alpha \rightarrow \beta &=_{df} \langle S, I, \sigma, t\rangle \not\models \alpha \text{ or } \langle S, I, \sigma, t\rangle \models \beta \\
\langle S, I, \sigma, t\rangle \models \alpha \leftrightarrow \beta &=_{df} \langle S, I, \sigma, t\rangle \models \alpha \rightarrow \beta \text{ and } \langle S, I, \sigma, t\rangle \models \beta \rightarrow \alpha \\
\langle S, I, \sigma, t\rangle \models \diamondminus \alpha &=_{df} \langle S, I, \sigma, t'\rangle \models \alpha \text{ for some } t' \text{ such that } t' < t
\end{aligned}
$$

Given a class of models κ, we say that a formula α has a *κ-model* or is *κ-satisfiable* if there exists $\langle S, I, \sigma, t\rangle \in \kappa$ such that $\langle S, I, \sigma, t\rangle \models \alpha$. We say that α is *κ-valid* if $\langle S, I, \sigma, t\rangle \models \alpha$ for all $\langle S, I, \sigma, t\rangle \in \kappa$. This is written $\models_\kappa \alpha$. When κ is clear from context or when κ is the class of all models we drop explicit reference to it in these expressions. □

2.3 Models of Computation

We will not be looking at the class of all models for purposes of protocol analysis. We now set out the class we will be using. We begin by describing some of the technical machinery we need. Our description of states and actions is motivated primarily by the formalisms operated on by the NRL Protocol Analyzer.

The model used by the Protocol Analyzer is an extension of the Dolev-Yao model [8]. We assume that the participants in the protocol are communicating in a network under the control of a hostile intruder who may also have access to the network as a legitimate user or users. The intruder has the ability to read all message traffic, destroy and alter messages, and create his own messages. Since all messages pass through the intruder's domain, any message that an honest participant sees can be assumed to originate from the intruder. Thus a protocol rule describes, not how one participant sends a message in response to another, but how the intruder manipulates the system to produce messages by causing principals to receive certain other messages.

As in Dolev-Yao, the words generated in the protocol obey a set of reduction rules (that is, rules for reducing words to simpler words), so we can think of the protocol as a machine by which the intruder produces words in the term-rewriting system. Also, as in Dolev-Yao, we make very strong assumptions about the knowledge gained when an intruder observes a message. We assume that the intruder learns the complete significance of each message at the moment that it is observed. Thus, if the intruder sees a string of bits that is the result of encrypting a message from A to B with a session key belonging to A and B, he knows that is what it is, although he will not know either the message or the key if he has not observed them.

A specification in the Protocol Analyzer describes how one moves from one state to another via honest participants sending data, honest participants receiving data, honest participants manipulating stored data, and the intruder's manipulation of data sent by the honest participants. Dishonest participants are identified with the intruder, and so are not modeled separately. The sending and receipt of messages by the intruder is not modeled separately, since it is automatically assumed that any message sent is received by the intruder, and any message received is sent by the intruder, even if it is only passed on by the intruder unchanged. Thus every receipt of a message by an honest principal implies the sending of a message by the intruder, and every sending of a message by an honest principal implies the receipt of a message by the intruder.

Given this, we look at the notion of a state more closely. One of the primary components of a state is a learned fact. Each honest protocol participant possesses a set of learned facts. Each learned fact is relevant to a given round of the protocol. A learned fact is described using an *lfact function*, which has four arguments. The first identifies the participant A for whom it is a learned fact. This will give us the agent of an action. The second identifies the round of the protocol via a round number that is local to the principal denoted by the first argument. This will allow each principal to attach each relevant action to a particular round of a particular protocol. The third indicates the nature of the fact. Generally this will indicate the action that the agent is taking. The fourth gives the present value of A's counter. In effect, this gives us a local clock value. The value of the lfact is either a list of words that make up the content of the fact, or if the fact does not have any content, it is [ ], the empty list.

One way we represent actions semantically is via changes in learned facts; however, we do not allow arbitrary changes in the value of lfact. A nonempty list can be the value of lfact for a given principal, round, and action, at the principal's local time T only if the value of lfact for that principal, round, and action, at the time immediately prior to T was [ ].

Thus, for example, suppose that A has attempted to initiate a conversation with B during local round N at time T. This can be expressed by the action (s, s') where the difference between s and s' is that in s,

$$\text{lfact(user(A,honest),N,init\_conv,T)} = [\,]$$

and in s',

$$\text{lfact(user(A,honest),N,init\_conv,T+1)} = [\text{user(B)}]$$

At any time prior to T, the value of the lfact would also be [ ].

It is also useful to allow certain actions to be 'forgotten'. This is accomplished by having a transition in which the value of lfact goes from a nonempty list to [ ].

Another component of a state is the intruder's knowledge, represented as a monotonically nondecreasing function of time. It is necessary to represent this in a manner distinct from the learned facts because the Analyzer represents the intruder in a different way than it represents ordinary principals. There are two kinds of actions associated with intruder knowledge that we allow. In the first of these, the intruder learns some word, that is, a string of symbols. For instance, suppose that A sends a message W to B at time $t1$, and the intruder intercepts (and thus learns) W at time $t2$. According to what we have set out above, this can be represented by (s, s'), where in s,

$$\text{lfact(user(A),N,send\_to\_B,T)} = [\,]$$

and in s',

$$\text{lfact(user(A),N,send\_to\_B,T+1)} = [\text{W}]$$

Then, the intruder learning of this action is given by (s', s''), where the only change from s' to s'' is that in s'' we have

$$\text{intruderknows(t2)} = \text{intruderknows(t1)} \cup \{[W]\}$$

where intruderknows(t) is the set of words known by the intruder at the global time t, and t1 and t2 are the global times corresponding to A's local times T and T + 1, respectively.

The second way the intruder may increase his knowledge is by performing some available internal operations on things he already knows. In other words, assuming ω is some n-ary operation of which the intruder is capable, if $\{W_1, \ldots, W_n\} \subseteq$ intruderknows(t), then

$$\text{intruderknows(t2)} = \text{intruderknows(t1)} \cup \{\omega(W_1, \ldots, W_n)\},$$

where t1 and t2 are again global times.

Definition 2.2 The four types of actions just given will be called '*basic actions*'. A *basic model* is one in which, for any given trace σ, one basic action may occur per unit time, and these specify the only allowable differences between a state and its successor. □

While basic models provide us with a simple model of computation in which to interpret the expressions of our language, they are too simple to be practical in most cases, especially as a basis for analysis using the NRL Protocol Analyzer. What we would like is a model in which state transitions can be complex enough to be useful but simple enough to provide assurance that our model is a reasonable one. To this end we introduce compressed models.

Definition 2.3 A *compressed model* is a model $\mathcal{M}$ for which there exists a basic model $\mathcal{M}'$ satisfying the following:

- The state space and interpretation for $\mathcal{M}$ and $\mathcal{M}'$ is the same.
- The trace σ in $\mathcal{M}$ is a subtrace of σ', the trace in $\mathcal{M}'$.

□

In particular, this means that for every transition (s_t, s_{t+1}) in σ, there exists a subsequence of σ', $(\sigma'(i), \ldots, \sigma'(i+n))$, such that $s_t = \sigma'(i)$ and $s_{t+1} = \sigma'(i+n)$. Note that in a compressed model a learned fact can change from one nontrivial value to another.

Now that we have the essentials of our semantics worked out, we can give a broad overview of how it functions in evaluating whether a protocol meets a set of requirements. For example, suppose we have a set of requirements describing one principal accepting a message from another, represented in our language by $\texttt{accept}(A, B, Mes, N)$. We can give a very simple description of the computational truth conditions for this action. For example, $\langle S, I, \sigma, t\rangle \models \texttt{accept}(A, B, Mes, N)$ iff in s_t lfact(user(A), N, accept_from_B, T) = [ ] and in s_{t+1} lfact(user(A), N, accept_from_B, T+1) = [Mes]. This is of course not very revealing. While what constitutes a send or receive action should be immediately clear, an `accept` action is somewhat complex. Thus, while we can present a model with such a simple interpretation, we need to give a more detailed interpretation of an `accept` action if we are to get any use out of it.

It would be hopeless to give general truth conditions for an `accept` action. Fortunately, at this point we can turn to the protocol in question to see what would constitute a reasonable interpretation of accepting a message. Accepting a message is what occurs when all the relevant checks have been verified by the accepting principal. Of course the atomic actions and their interpretations can be quite different when we move to an entirely distinct class of protocols, e.g., resource sharing protocols. The exact details of how this works will be set out below when we describe how to specify a protocol for the Analyzer.

Once we have set an interpretation for all of the expressions used in the statement of requirements and have specified the protocol itself, we are in a position to determine whether or not the protocol meets the requirements. Given a fixed state space S and interpretation I, we consider the class κ of all models $\langle S, I, \sigma, t\rangle$ for which σ is a trace of the protocol specification. To see if the protocol meets the requirements we simply see if the formulae that constitute the requirements are valid in κ. Of course, while the check is very simple in theory, it is rather difficult in practice. This is where the NRL Protocol Analyzer comes in: it helps us to make the determination. That is, to see if the protocol meets the requirements we present the Analyzer with the requirements and the interpretation of atomic actions therein. We then ask it to determine if the models in κ are a subclass of those that make the requirements true. We will show how to do this below for a sample protocol and sample set of requirements. The analysis is primarily conducted in the language of the Analyzer, which for us amounts to a semantic description language. Thus, we present a description of the Analyzer and its language before proceeding further.

3 The NRL Protocol Analyzer

3.1 The Specification Language Used by the NRL Protocol Analyzer

A specification in the NRL Protocol Analyzer consists of four sections. The first section consists of transition rules governing the actions of honest principals. It may also contain rules describing possible system failures that are not necessarily the result of actions of the intruder, for example, the compromise of a session key. The second section describes the operations that are available to the honest principals and possibly to the intruder, e.g., encryption and decryption. The third section describes the atoms that are used as the basic building blocks of the words in the protocol. The fourth section describes the rewrite rules obeyed by the operations.

A transition rule has three parts. The first part gives the conditions that must hold before the rule can fire. These conditions describe the words the intruder must know (that is, the message that must be received by the principal), the values of the lfacts available to the principal, and any constraints on the lfacts and words. At the moment, the syntax of the constraints on words is somewhat restricted; they can only say that words must or must not be of a given length or that they must or must not be equal to other words. The second part describes the conditions that hold after the rule fires in terms of words learned by the intruder (that is, the message sent by the principal) and any new values taken on by lfacts. Each time a rule fires, the principal's local time is incremented; this is also recorded in the preconditions and postconditions of the rule. The third part of the rule consists of an event statement. It is used to record the firing of a rule and is useful for indicating what the rule does. It is derived from the first two parts of the rule. The event statement describes a function with four arguments. The first gives the name of the relevant principal. The second gives the number of the protocol round. The third identifies the event. The fourth gives the value of the principal's counter after the rule fires. The value of the event is a list of words relevant to the event.

As an example, consider the initial session request that A sends in the Kerberos protocol, namely, A sends to S: A, B. The corresponding rule is as follows:

```
If:
count(user(A,honest)) = [N],
then:
count(user(A,honest)) = [s(N)],
intruderlearns([user(A,honest),user(B,Y)]),
lfact(user(A,honest),N,init_sendsrequest,s(N)) =
[user(B,Y)],
EVENT:
event(user(A,honest),N,init_sendsrequest,s(N)) =
      [user(B,Y)].
```

There are no conditions for this rule to fire other than to identify the local time. We also use the local time at the start of the protocol for the local round number of the protocol run. Although the intruder is generally assumed to know the names of all principals, we reflect what A sent via the intruderlearns postcondition. The initsendrequest lfact indicates that it is user(B,Y) with whom user(A,honest) wishes to have a session key. This is also reflected in the event statement. In this rule B is not indicated to be honest or dishonest since that is unrelated to this transition.

The second section of the specification defines the operations that can be made by honest principals and by the intruder. If an operation can be made by the intruder, the Analyzer translates it into a transition rule similar to the above, except that the relevant principal is the intruder instead of an honest principal, and no lfacts are involved. An example of a specification of an operation is the following, describing public key encryption:

```
fsd1:pke(X,Y):length(X) = 1:
     length(pke(X,Y)) = length(Y):pen.
```

The term "fsd" stands for "function symbol description." The next term gives the operation and the arguments. The third gives conditions on the arguments. In this case, we make the condition that the key be a certain length, which in this case we make a default unit length one. The next term gives the length of the resulting word, which in this case is the length of Y. The last field is set to "pen" if we are assuming that the penetrator can perform the operation, and "nopen" if we are assuming that he can't. Thus the decision to put "pen" or "nopen" into the last field may vary with our assumptions about the environment in which the protocol is operating.

Some operations are built into the system. These are: concatenation, taking the head of a list, taking the tail of a list, and id_check, which is used by an honest principal to determine whether or not two words are identical.

The third section describes the words that make up the basic building blocks. We call these words "atoms". Examples would be user names, keys, and random numbers. Again, we indicate whether or not the word is known to the intruder in the last field of an atom specification, it is "known" if the intruder knows it initally, and "notknown" if the intruder doesn't know it initially.

The last section describes the rewrite rules by which words reduce to simpler words. An example of a rewrite rule would be one which describes the fact encryption with corresponding public and private keys cancel each other out:

```
rr1: pke(privkey(X),pke(pubkey(X),Y)) => Y.
rr2: pke(pubkey(X),pke(privkey(X),Y)) => Y.
```

One queries the Analyzer by asking it to find a state that consists of a set of words known by the intruder, a set of lfacts, and a sequence of events that must have occurred. One can put conditions on the words, lfacts and events by putting conditions on the words that appear in them. One can also put conditions on the results by specifying that certain sequences of events must *not* have occurred.

The Analyzer then matches up the output of each rule with the specified state, if possible, by performing substitutions on the output that makes it reducible, via the reduction rules, to the state specified. It may match either the entire state or some subset. The input of the rule together with any part of the state then becomes a new state to be queried.

The way in which the Analyzer interprets rules allows considerable freedom in how the matching is done. Variables are local to rules, and, each time a rule is applied, a new set of variables is generated. This allows the Analyzer, for example, to develop scenarios involving multiple instantations of protocol rounds, as well scenarios in which the same principal plays more than one role.

4 Two Party Key Agreement

In section 1.1 above, we gave an example set of requirements for two party, one server key distribution. In this section we consider requirements where the session key is not distributed from a single server. Rather parts of it are produced be each of two principals. This is generally called key agreement rather than key distribution because the session key itself need never be sent. Each principal can send an agreement key to the other, and the session key is a function of the two agreement keys. Despite these differences there is a large amount of overlap between the requirements of section 1.1 and those we now give. In fact the first two are the same. (The only nominal difference is that in the following section we use '`init_accept_sk`' to indicate the intiator's acceptance of a session key and '`init_accept_ak`' for the initiator's acceptance of an agreement key, and we follow similar usage for the receiver.)

4.1 Two Party Key Agreement Requirements

Requirement 1 If a key has been accepted, it should not be learned by the intruder, except through a compromise event:

$$\begin{aligned}&\Diamond(\texttt{init\_accept\_sk}(user(A, honest), user(B, honest), K, M1)\vee\\&\qquad\texttt{rec\_accept\_sk}(user(B, honest), user(A, honest), K, M2)) \rightarrow\\&\Diamond(\texttt{learn}(pen, (), K, M?) \rightarrow \Diamond\texttt{compromise}(environment, (), K, M?))\end{aligned}$$

Requirement 2 If a key is accepted for communication between two honest parties, it should not have been accepted in the past, except by the other party. This becomes two requirements, one for the initiator and one for the receiver. (The receiver's requirement is a mirror image of the sender's.)

$$\begin{aligned}&\texttt{init\_accept\_sk}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), K, M1) \rightarrow\\ &\quad\neg(\diamondsuit\texttt{init\_accept\_sk}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?)\wedge\\ &\quad(\diamondsuit\texttt{rec\_accept\_sk}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?) \rightarrow (C = B \wedge D = A))\end{aligned}$$

$$\begin{aligned}&\texttt{rec\_accept\_sk}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), K, M2) \rightarrow\\ &\quad\neg(\diamondsuit\texttt{rec\_accept\_sk}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?)\wedge\\ &\quad(\diamondsuit\texttt{init\_accept\_sk}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?) \rightarrow (C = A \wedge D = B))\end{aligned}$$

Requirement 3 If an agreement key is accepted by either party for a communication session between two parties, then that key must have been previously sent by the other party. (This requirement introduces the notation 'AK_P' for principal P's agreement key, i.e., P's contribution to the session key.)

$$\begin{aligned}&\texttt{init\_accept\_ak}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), AK_B, M?) \rightarrow\\ &\qquad\diamondsuit(\texttt{send}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), AK_B, M?)\end{aligned}$$

$$\begin{aligned}&\texttt{rec\_accept\_ak}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), AK_A, M?) \rightarrow\\ &\qquad\diamondsuit(\texttt{send}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), AK_A, M?)\end{aligned}$$

Requirement 4 If a session key is accepted by either party for a communication session between two parties, then that party must have previously generated her own agreement key and accepted the other principal's agreement key, and the session key must bear the correct functional relationship to the agreement keys. (This requirement introduces the notation 'f' for the key agreement function used.)

$$\begin{aligned}&\texttt{init\_accept\_sk}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), K, M?) \rightarrow\\ &\qquad(\texttt{init\_accept\_ak}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), AK_B, M?)\wedge\\ &\qquad\qquad(\texttt{generate}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), AK_A, M?))\wedge\\ &\qquad K = f(AK_A, AK_B)\end{aligned}$$

$$\begin{aligned}&\texttt{rec\_accept\_sk}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), K, M?) \rightarrow\\ &\qquad(\texttt{rec\_accept\_ak}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), AK_A, M?)\wedge\\ &\qquad\qquad(\texttt{generate}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), AK_B, M?))\wedge\\ &\qquad K = f(AK_A, AK_B)\end{aligned}$$

Note that we omit the $\diamondsuit$. This is because, even though an event such as generating the agreement key must logically precede the generation of the session key, in a protocol specification we may specify both as part of the same state transition. They will thus appear to be occuring simultaneously. However, we can specify that, if the session key was generated, then so was the agreement key.

The most familiar examples of key agreement are probably implementations of Diffie-Hellman key exchange [7]. It may not be obvious how the above requirements apply to Diffie-Hellman, in which principals exchange public agreement keys and the session key is a function of either public agreement key with the other party's private agreement key. But, in Diffie-Hellman key exchange the public agreement keys are in effect encrypted versions of the private agreement keys. If we represent P's public agreement key as '$pub(AK_P)$', then the key agreement function f is implicitly defined via an explicit function g such that $g(AK_A, pub(AK_B)) = g(AK_B, pub(AK_A)) = f(AK_A, AK_B)$. Diffie-Hellman satisfies further requirements, such as keeping each principal's (private) agreement key from being known by the other. This is a useful requirement that can be additionally specified if required; however, since many agreement algorithms are not intended to meet such a requirement, we do not set it out here.

Notice that there is very little in the above requirements that guarantees that keys are recently generated, unused, etc. The only requirement at all in this general area is requirement 2, which basically rules out the acceptance of previously accepted session keys. One reason for this situation is that such requirements are not as generally desirable for key agreement protocols as they are for key distribution protocols. As always, it should be recalled that the requirements we set out are meant to be generic for a class; some requirements may be unnecessary to meet all the goals for some protocols in the class, indeed it is possible that some of those goals be inconsistent with some of the generic requirements. The following are all properties that we might want keys to have, and which we take to be the primary ones in this area. *Freshness* is the property of having been generated or sent after the start of the present epoch. *Currency* is the property of being part of the present protocol run. (Currency usually implies freshness, but it may not.) *Virginity* is the property of not having been accepted for use previously (accept possibly by other principals in the protocol). So, if we wanted to require that the receiver's agreement key be virgin we would formalize this:

Requirement 5

$$\begin{aligned}
&\texttt{init\_accept\_ak}(user(A, honest), user(B, honest), AK_B, M?) \rightarrow \\
&\quad \neg\Diamond(\texttt{init\_accept\_ak}(user(C, honest), user(D, X), AK_B, M?)\vee \\
&\qquad \texttt{rec\_accept\_ak}(user(C1, honest), user(D1, X), AK_B, M?))
\end{aligned}$$

Even if we require that the session key have all of the above properties, this is generally met if either of the agreement keys has all of them; there is no guarantee that both of the agreement keys have them.[3] Thus, the need for requirements concerning these properties may be subtle. As an example, consider ElGamal's one pass variant on basic Diffie-Hellman [9]. In this protocol the receiver's public agreement key is expected to be available prior to the protocol run. The run consists of a single message in which the initiator sends his public agreement key to the receiver. Thus, in this protocol the initiator's agreement key is fresh, current, and virgin; therefore, so is the session key. But, the receiver's key is not current, and it need not be either virgin or fresh.

In order to demonstrate the application of our requirements language and the NRL Protocol Analyzer we now look at a specific key agreement protocol to see how it meets the above requirements. The application turned out to be illuminating for several reasons. First, it made clear an underlying assumption that was necessary to the correct operation of the protocol. Secondly, it pointed out some situations in which our key

[3] In [18] and [20] we discuss key distribution protocols where the session key is accepted more than once.

requirements of freshness, currency, and virginity may not be adequate. We outline the protocol specification and analysis in detail below.

4.2 A Protocol Analysis Example

In [3], Aziz and Diffie present the design of "a secure communication protocol that provides for both privacy of wireless data communications and authenticity of the communicating parties." This protocol relies on input from both principals to form the session key. It is thus reasonable to analyze this protocol to see if it satisfies the above requirements. The protocol runs as follows, where A is a mobile unit and B is a base unit.

4.2.1 The Aziz Diffie Protocol

(1) A *sends to* B: $Cert_A, N_A, alglist_A$
(2) B *sends to* A: $Cert_B, \{AK_B\}_{K_A}, algchoice_B, \{md(\{AK_B\}_{K_A}, algchoice_B, N_A, alglist_A)\}_{K_B^{-1}}$
(3) A *sends to* B: $\{AK_A\}_{K_B}, \{md(\{AK_A\}_{K_B}, \{AK_B\}_{K_A})\}_{K_A^{-1}}$

$Cert_X$ is a certificate signed by a trusted authority containing X's public identity, public key and other information. $alglist_A$ is a list that A has composed of encryption algorithms, and $algchoice_B$ is an algorithm chosen from $alglist_A$ by B. The shared key is the exclusive-or of AK_A and AK_B, that is A's and B's agreement keys respectively. The term "md(X)" stands for a collision-free message digest function computed over X.

When B receives A's message, he verifies A's certificate, and chooses an algorithm from the list of encryption algorithms. He generates an agreement key AK_B, encrypts it with A's public key and sends it to A. This encrypted agreement key will serve double duty as a nonce. The encrypted key, the chosen algorithm, and A's nonce, are signed with B's private key and sent to A together with the certificate.

When A receives B's message, she verifies the signature and verifies the message's recency by checking the nonce. She then decrypts the encrypted key and accepts that as B's key agreement key. She then generates an agreement key AK_A, encrypts it with B's key, and sends this to B together with her signature computed over hers and B's encrypted key agreement keys.

When B receives the message, he verifies that the signature is computed over the two agreement keys. This verifies the recency of the key and the fact that it comes from A. B then accepts AK_A as A's agreement key.

4.2.2 Specification of the Aziz-Diffie Protocol

In this section we describe how we specified the Aziz-Diffie protocol in the Protocol Analyzer language.

Specification of Principals and Words Used in the Protocol We begin by specifying the principals involved in the protocol. Initiators of the protocol are always mobile units, and receivers are always base units. This differs from most other key distribution protocols, in which the same principal may play either role. It thus rules out some multiple-role attacks that rely on the interleaving of two or more protocols in which the same principal plays different roles.

Both mobile and base units can be either honest or dishonest. Honest units obey the rules of the protocol, and do not share information with the intruder except when it is sent in a message. Dishonest units are assumed to be in league with the intruder and to share all information learned with the intruder. We thus designate a mobile unit as mobile(A,H) and a base unit as base(A,H), where A is a variable uniquely identifying a principal, and H is a variable which can be set either to "honest" or "dishonest".

We also specify a principal we call "dummy." The purpose of the dummy is to aid in specifying key compromise. Since the Analyzer specifies a protocol as a transition on state variables, we need a place to hold the key as a state variable before it is compromised. It is not appropriate to hold it as a state variable belonging to one of the relevant principals, since a key may be compromised long after it is deleted from the memory of the relevant principals. Thus the dummy is created as a place to hold a key until it is compromised.

Random numbers and agreement keys are written as function symbols with name and time as arguments. The reason for this is that the Protocol Analyzer treats terms as equal if and only if they are syntactically equal. Thus attaching a name-time stamp to nonces and keys captures the assumption that keys and nonces generated either at different times or by different principals are different. A key generated by U at time N is designated by key(U,N). A random number generated by U at time N is designated by rand(U,N).

Public and private keys are identified by their owners. The term pubkey(U) stands for U's public key, and the term privkey(U) stands for U's private key.

We also develop conventions for algorithms, lists of algorithms, serial numbers and so forth in a similar way. If only one version of a word is associated with a particular principal, the word is described as a function of a name. If the principal can generate different versions of a word at different times, the word is described as a function of name and time.

Specification of Operations and Rewrite Rules We define three operations: public-key encryption of Y with key X, denoted by pke(X,Y), message digest computed over X, denoted by md(X), and exclusive-or of X and Y, denoted by xor(X,Y). There are also several built-in operations: concatenation, (X,Y), head and tail of list, head(X) and tail(X), and id_check(X,Y), which denotes the result of checking if X and Y are identical.

The rewrite rules for public key encryption and decryption are specified as follows:

```
rr(1): pke(privkey(X),pke(pubkey(X),Y)) => Y.
rr(2): pke(pubkey(X),pke(privkey(X),Y)) => Y.
```

The Protocol Analyzer does not yet handle commutative rules, so we could give a complete specification of the properties of exclusive-or. We included only one rewrite rule describing a cancellation property:

```
rr(3): xor(X,xor(X,Y)) => Y.
```

The incomplete specification of the properties of exclusive-or means that we should not consider our analysis of the Aziz-Diffie protocol as a complete one. Even so, we were able to discover some interesting properties.

The built-in functions also obey rewrite rules. For example, id_check(X,X) reduces to "ok."

Specification of Words Known Initially The intruder is assumed to know all principal names, all public keys, all private keys belonging to dishonest principals, and all random numbers and session keys generated by dishonest principals. We also found it useful to specify words such as the public key certificates that are easy for the intruder to learn as words known initially. This saves the Analyzer the work of having to recreate the path by which the intruder learns the word each time it is desired.

Specification of the Protocol Transitions The transitions specify the actions of the honest principals. However, we note that in each case a principal may be dealing with an honest or dishonest principal.

In the first transition, an honest mobile unit mobile(A,honest) sends an initiation message to a base unit base(B,H) who may be honest or dishonest. This is specified as follows:

```
rule(1)
If:
count(mobile(A,honest)) = [N],
then:
count(mobile(A,honest)) = [s(N)],
intruderlearns([(serial(mobile(A,honest)),
validity(mobile(A,honest)),mobile(A,honest),pubkey(mobile(A,honest)),
keyauth),
pke(privkey(keyauth),md((serial(mobile(A,honest)),
validity(mobile(A,honest)),mobile(A,honest),pubkey(mobile(A,honest)),
keyauth))), rand(mobile(A,honest),N),alglist]),
lfact(mobile(A,honest),N,init_request,s(N)) = [base(B,H),rand(mobile(A,honest),N)],
EVENT:
event(mobile(A,honest),N,init_keyrequest,s(N)) =
   [base(B,H),rand(mobile(A,honest),N)].
```

The mobile unit remembers the random number it generated, and who it was trying to talk to, and stores this in the variable init_request.

The next two transitions describes an honest base unit's response to a message purporting to come from an honest or dishonest mobile unit. In the first transition, base(B,honest) receives the message and sets up the verification procedures, in which it performs an identity check (id_check) on the result of computing the md function on the mobile unit's public key certificate and applying the authorizing entity's public key to the signature on the certificate. In the second transition, base(B,honest) performs the verification, and, if it is successful, sends a return message to the mobile(A,H).

This is specified as follows.

```
rule(2)
If:
count(base(B,honest)) = [N],
intruderknows([X,Y,R,alglist]),
then:
count(base(B,honest)) = [s(N)],
lfact(base(B,honest),N,rec_nonce,s(N)) =
   [head(tail(tail(X))),head(tail(tail(tail(X)))),R,
    id_check(md(X),pke(pubkey(keyauth),Y))],
EVENT:
event(base(B,honest),N,rec_initmess,s(N)) =
   [X,Y,R].
```

```
rule(3)
If:
count(base(B,honest)) = [N],
lfact(base(B,honest),M,rec_nonce,N) = [U,Key,R,ok],
then:
count(base(B,honest)) = [s(N)],
intruderlearns([(serial(base(B,honest)),
validity(base(B,honest)),base(B,honest),pubkey(base(B,honest)),
keyauth),
pke(privkey(keyauth),md((serial(base(B,honest)),
validity(base(B,honest)),base(B,honest),pubkey(base(B,honest)),
keyauth))),
pke(Key,key(base(B,honest),N)),
algtype(base(B,honest)),
pke(privkey(base(B,honest)),
   md((pke(Key,key(base(B,honest),N)),
   algtype(base(B,honest)),
   R,
   alglist)))]),
lfact(base(B,honest),M,rec_seskeyhalf1,s(N)) = [U,Key,key(base(B,honest),N)],

EVENT:
event(base(B,honest),M,rec_genkey,s(N)) = [U,Key,R,key(base(B,honest),N)].
```

In the next two transitions, the honest mobile unit receives the message purporting to come from the base unit it is trying to communicate with. In the first transition, it receives the message and performs three verifications. In the first, it verifies the base unit's certificate. In the third, it verifies the base unit's signature on the message. The second verification is not quite so obvious, however. The mobile unit verifies that the result of decrypting the word Randnum is of the form key(U,J). What it is doing is a format check; it is verifying that the result of a decryption is properly formatted as a key. In other words, if Randnum was something other than the result of encrypting a key with the mobile unit's public key, the check would fail. This format check turns out to be necessary to prevent a particular kind of denial of service attack. It is not specified in the protocol, but, upon contacting the authors, we found that this was an assumption that was being made. We discuss the attack and the conditions under which it can be successful in more detail in the section on the analysis of the protocol.

In the next transition, if the verification succeeds, the mobile unit accepts the base unit's agreement key and generates the session key using that and its own agreement key. The key is also transferred to the dummy principal, where it is stored in the variable dummy_vulnerable. This will be used to model key compromise in a later transition.

```
rule(4)
If:
count(mobile(A,honest)) = [N],
intruderknows([Cert,Certsig,Randnum,Algtype,Sigrand]),
lfact(mobile(A,honest),M,init_request,N) =
   [B,R],
then:
count(mobile(A,honest)) = [s(N)],
lfact(mobile(A,honest),M,init_baseinfo,s(N)) =
   [head(tail(tail(Cert))),head(tail(tail(tail(Cert)))),
id_check(md(Cert),pke(pubkey(keyauth),Certsig))],
lfact(mobile(A,honest),M,init_randinfo,s(N)) =
[Randnum,pke(privkey(mobile(A,honest)),Randnum),
  id_check(pke(privkey(mobile(A,honest)),Randnum),key(U,J)),
  id_check(md((Randnum,Algtype,R,alglist)),
  pke(head(tail(tail(tail(Cert)))),Sigrand))],
EVENT:
event(mobile(A,honest),M,init_gotkey,s(N)) =
[B,R,Cert,Certsig,Randnum,Algtype,Sigrand].
```

```
rule(5)
If:
count(mobile(A,honest)) = [N],
count(dummy) = [N1],
lfact(mobile(A,honest),M,init_request,N) = [B,R],
lfact(mobile(A,honest),M,init_baseinfo,N) = [B,Bkey,ok],
lfact(mobile(A,honest),M,init_randinfo,N) = [Rand1,BK,ok,ok],
then:
count(mobile(A,honest)) = [s(N)],
count(dummy) = [s(N1)],
lfact(mobile(A,honest),M,init_keyhalf,s(N)) = [key(mobile(A,honest),N)],
lfact(mobile(A,honest),M,init_key,s(N)) =
   [B,xor(BK,key(mobile(A,honest),N))],
lfact(dummy,N1,dummy_vulnerable,s(N1)) =
   [xor(BK,key(mobile(A,honest),N))],
intruderlearns([pke(Bkey,key(mobile(A,honest),N)),
pke(privkey(mobile(A,honest)),
  md((pke(Bkey,key(mobile(A,honest),N)),Rand1)))]),
EVENT:
event(mobile(A,honest),M,init_acceptkey,s(N)) =
[B,R,Bkey,Rand1,BK,key(mobile(A,honest),N)].
```

The next two transitions define the base unit's behavior upon receiving the last message of the protocol. In the first transition, it checks the format of the encrypted key upon decryption with the base unit's private key, and the moble unit's signature on the mobile unit's and the base unit's encrypted agreement keys. In the next transition, if the check works out, the base unit generates the session key, and the key is transferred to the dummy principal.

```
rule(6)
If:
count(base(B,honest)) = [N],
lfact(base(B,honest),M,rec_seskeyhalf1,N) = [A,Pubkey,Key1],
intruderknows([Rand2,Sigrand2]),
then:
count(base(B,honest)) = [s(N)],
lfact(base(B,honest),M,rec_testkey,s(N)) =
[A,Key1,pke(privkey(base(B,honest)),Rand2),
id_check(pke(privkey(base(B,honest)),Rand2),key(U,J)),
id_check(md((Rand2,pke(Pubkey,Key1))),pke(Pubkey,Sigrand2))],
EVENT:
event(base(B,honest),M,rec_gotkey,s(N)) = [A,Pubkey,Key1,Rand2,Sigrand2].



rule(7)
If:
count(base(B,honest)) = [N],
count(dummy) = [N1],
lfact(base(B,honest),M,rec_testkey,N) = [A,Key1,AK,ok,ok],
then:
count(base(B,honest)) = [s(N)],
count(dummy) = [s(N1)]
lfact(base(B,honest),M,rec_goodkey,s(N)) = [xor(Key1,AK)],
lfact(dummy,N1,dummy_vulnerable,s(N1)) = [xor(Key1,AK)],
EVENT:
event(base(B,honest),M,rec_acceptskey,s(N)) = [A,Key1,AK].
```

The last transition describes the transition by which a key is compromised. If a word resides in a dummy_vulnerable variable, then it can be learned by the intruder if this transition fires.

```
rule(10)
If:
count(dummy) = [N],
lfact(dummy,M,dummy_vulnerable,N) = [K],
then:
count(dummy) = [s(N)],
intruderlearns([K]),
lfact(dummy,M,dummy_vulnerable,s(N)) = [],
EVENT:
event(dummy,M,dummy_compromised,s(N)) = [K].
```

4.2.3 Mapping the Requirements to the Specification

In this section we describe how the statements in the requirements for two-party key agreement protocols can be mapped to the protocol so that they can be verified using the Protocol Analyzer.

When we present a query to the Analyzer, we have several options. We can ask it to find a set of lfact values, a set of words the intruder knows, a sequence of events that occurred, or some combination of the above. We can also put conditions on the results it finds. We can require that words have certains properties, and require that certain sequences of events do *not* occur.

We use the Analyzer to attempt to prove a state is unreachable. Thus we must translate each requirement into a description of an unreachable state. This is done in two parts. First, each requirement R is translated into an equivalent requirement of the form not(R'). Secondly, the actions described in the requirement are translated into the corresponding event statement used by the Analyzer, transforming R' into a state description R'' that can be presented to the Analyzer. The Analyzer is then used to prove R'' unreachable. If this can be done, it has been proven that the requirement holds.

We begin by mapping action statements that describe actions of honest principals to event statements. Again, we note that this mapping depends on the context of the protocol. We describe the mappings of all action statements, except the intruder_learns statement, in the table below. In this table n_a denotes the action statement, and n_b denotes the corresponding event statement.

1_a	init_accept_sk$(A, B, xor(AK_A, AK_B), M)$
1_b	event(A, M, init_acceptkey, N) = [B, X1, X2, X3, AK_B, AK_A]
2_a	rec_accept_sk$(B, A, xor(AK_A, AK_B), M)$
2_b	event(B, M, rec_acceptskey, N) = [A, AK_A, AK_B]
3_a	init_accept_ak(A, B, AK_B, M)
3_b	event(A, M, init_acceptkey, N) = [B, X1, X2, X3, AK_B, X4]
4_a	rec_accept_ak(B, A, AK_B, M)
4_b	event(B, M, rec_acceptskey, N) = [A, X1, AK_B]
5_a	init_generate(A, B, AK_A, M)
5_b	event(A, M, init_acceptkey, N) = [B, X1, X2, X3, X4, AK_A]
6_a	rec_generate(B, A, AK_B, M)
6_b	event(B, M, rec_genkey, N) = [A, X1, X2, AK_B]
7_a	request$(A, B, (), M)$
7_b	event(A, M, init_keyrequest, N) = [B, X1]
8_a	compromise$(environment, (), K, M)$
8_b	event(dummy, M, dummy_compromise, N) = [K]

Note that several action statements can map to the same event statement. The event statement init_acceptkey has no less than three action statements mapping to it, one describing acceptance of a session key, one describing acceptance of an agreement key, and one describing generation of an agreement key. It is also possible that an action statement can map to more than one event statement, as we shall see in the case of the intruder_learns action statement.

Mapping intruder actions to the protocol specification is trickier, since intruder actions, which consist of learning words, do not map to specific transitions, but instead to any transition which can produce a word of the appropriate form. However, we recall that, in querying the Analyzer, we can ask it to produce a state in which the intruder knows a word or words. This corresponds to asking it to find a state in which the intruder learned that word in the past. If all we wish to prove is that the intruder learned that word in the past, and we are not concerned about ordering, then this is sufficient. In most cases, we are mainly concerned with proving that an intruder never learns a word; for example, we want to prove that the intruder never learns a key, not that he does not learn it before or after it is used. Thus, in most cases, the way in which the Analyzer is queried will be sufficient.

If order is important, we can change the way in which a protocol is modeled so that the variety of events in which an intruder learns a word is reduced. Instead of having the intruder learn the work directly when a message is sent, we create an entity called "network," and let the words of the message be entered in one or several of network's state variables. The intruder's learning of a message could be represented in a way similar to the compromise transition as follows:

```
rule(9)
If:
count(network) = [N],
lfact(network,M,network_word(Q),N) = [X],
then:
count(network) = [s(N)],
intruderlearns([K]),
lfact(network,M,network_word(Q),s(N)) = [],
EVENT:
event(network,M,network_tointruder,s(N)) = [K].
```

This would not only reduce the number of event statements corresponding to the intruder_learns action statement, but also allow us to model situations where we do not assume that the intruder learns all messages that are sent over the network. However, it also adds complexity to the analysis, since an extra state transition is needed before the intruder learns a word from a message. For this reason, and since we did not have any requirements imposing conditions on the order in which the intruder learns a word, we chose not to model things this way for the Aziz-Diffie protocol.

We now show how we would present the various requirements to the Analyzer. The Analyzer is used by specifying an insecure state and showing that it is unreachable, so we specify the negation of the requirement and showing that it is unreachable. In the current version of the Analyzer, it is possible to present as goals states made up of the following:

1. A set of words known by the intruder;
2. A set of values of local state variables;
3. An sequence of events that occured; and
4. A set of sequences of events that must not have occurred.

A requirement that the occurrence of some event E implied that some other sequence of events S occurred would be proved by taking the negation of the requirement, that is, the requirement that E occurred and S did not occur, and proving that that is unreachable. This can easily be presented as a goal to the Analyzer, as we see from above. A requirement that the occurence of some event E implies that S occurred previously can be proved by showing that the state in which E occurred and (S,E) did not occur is unreachable, where (S,E) is the sequence obtained by appending E to S. A requirement that, if E occurred then S did not occur previously, can be proved by showing that the state in which the sequence (S,E) occured is unreachable. A requirement that if E occurred S occurred can be proved by showing two goals unreachable: one in which the sequence (S,E) occurred and one in which the sequence S' occurred, where S' is obtained by identifying the final event in S with E (if possible). Finally, a requirement that if S occurred, then the intruder did not learn a set of words W can be proved by showing that the state in which S occurred and in which the intruder knows W is unreachable. We have found these constructs adequate for all the protocol requirements we have generated so far.

We now give some examples in which requirements are mapped to a state in the Aziz-Diffie protocol specification. Consider Requirement 3, which says that, if an agreement key is accepted for communication with another party, then it must have been previously sent by the party. For the initiatior, this is given as the requirement

$$\begin{aligned}&\texttt{init\_accept\_ak}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), AK_B, M?) \rightarrow \\ &\quad \diamondminus(\texttt{send}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), AK_B, M?)\end{aligned}$$

From Table 1, we get that

$$\texttt{init\_accept\_ak}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), AK_B, M?)$$

maps to

$$\begin{aligned}E_1 = \texttt{event(mobile(A,honest),M1,init\_acceptkey,N1)} = \\ \texttt{[base(B,honest),X1,X2,X3,AK\_B,X4]}\end{aligned}$$

while

$$(\texttt{generate}(\mathit{user}(B, \mathit{honest}), \mathit{user}(A, \mathit{honest}), AK_B, M?)$$

maps to

$$E_2 = \texttt{event(base(B,honest),M2,rec\_genkey,N2)} = \texttt{[mobile(B,honest),X1,X2,AK\_B]}.$$

By our above argument, we need to show that the state in which the event E_1 occurred and (E_2,E_1) did not occur is unreachable.

As another example, consider Requirement 2, which says that, if a key is accepted for communication between two parties, it should not have been accepted in the past, except by the other party in the role of receiver. The requirement for the initiator is:

$$\begin{aligned}&\texttt{init\_accept\_sk}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), K, M1) \rightarrow \\ &\quad \neg(\diamondminus\texttt{init\_accept\_sk}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?)\wedge \\ &\quad (\diamondminus\texttt{rec\_accept\_sk}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?) \rightarrow (C = B \wedge D = A))\end{aligned}$$

In order to map this requirement to the Protocol Analyzer, we need to break it down into two requirements:

$$\begin{aligned}&\texttt{init\_accept\_sk}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), K, M1) \rightarrow \\ &\quad \neg(\diamondminus\texttt{init\_accept\_sk}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?)\end{aligned}$$

$$\begin{aligned}&\texttt{init\_accept\_sk}(\mathit{user}(A, \mathit{honest}), \mathit{user}(B, \mathit{honest}), K, M1) \rightarrow \\ &\quad (\diamondminus\texttt{rec\_accept\_sk}(\mathit{user}(C, \mathit{honest}), \mathit{user}(D, X), K, M?) \rightarrow (C = B \wedge D = A))\end{aligned}$$

In the first case, we must prove that the state in which the sequence of events (E_1,E_2) occurred is unreachable, where E_2 =

$$\texttt{event(mobile(A,honest),M,init\_acceptkey,N)} = [\texttt{base(B,honest),X1,X2,X3,AK\_B,AK\_A}]$$

and E_1 is the event

$$\texttt{event(mobile(C,honest),M,init\_acceptkey,N)} = [\texttt{D,Y1,Y2,Y3,AK\_B,AK\_A}].$$

.

In the second case, we must prove that the state in which the sequence of events (E_1,E_2) is unreachable, where E_2 is as above, and E_1 =

$$\texttt{event(base(C,honest),M,rec\_acceptskey,N)} = [\texttt{D,AK\_A,AK\_B}]$$

where either `base(C,honest)` is not equal to `base(B,honest)` or `D` is not equal to `mobile(A,honest)`.

4.2.4 Analyzing the Requirements

After modeling the negated requirements as goals for the NRL Protocol Analyzer, we ran the Analzyer to determine if any of the negated requirements were reachable. Although we did not discover any actual weaknesses in the protocol, we found one hidden assumption, and one case in which our requirements may be too lenient. We should note, however, that our analysis was not complete, since we did not model all the algebraic properties of exclusive-or.

The hidden assumption was discovered when we ran the Analyzer on Requirement 3 for the initiator using an earlier version of the specification, a version in which format checking was not specified. In that version we found an attack that ran as follows

1. *A sends to B* : $Cert_A, N_A, alglist_A$
   (intercepted by I)
2. I_C *sends to B*: $Cert_C, N_A, alglist_A$
3. *B sends to C*: $Cert_B, \{AK_B\}_{K_C}, algchoice_B, \{md(\{AK_B\}_{K_C}, algchoice_B, N_A, alglist_A)\}_{K_B^{-1}}$
   (intercepted by I)
4. I_B *sends to A* : $Cert_B, \{AK_B\}_{K_C}, algchoice_B, \{md(\{AK_B\}_{K_C}, algchoice_B, N_A, alglist_A)\}_{K_B^{-1}}$

A checks the signature, and applies her private key to $\{AK_B\}_{K_C}$ to obtain $\{\{AK_B\}_{K_C}\}_{K_A^{-1}}$, which she then thinks is the key from B. She then generates her own key agreement key, and accepts as the session key the result of taking the exclusive-or of it with $\{\{AK_B\}_{K_C}\}_{K_A^{-1}}$.

We note that this attack results in at worst a denial of service, since, although the intruder convinces A that a nonkey is a key, the intruder never learns the word that A accepts as a key, and thus cannot impersonate A or B or read encrypted traffic. It can also be easily foiled if the message B encrypts with A's public key is formatted in such a way that A can determine whether or not it is an actual message after A decrypts it. In that case, $\{\{AK_B\}_{K_C}\}_{K_A^{-1}}$ is unlikely to satisfy the formatting requirements, and A will reject it. It is reasonable to expect that such formatting will be used. However, by the use of the NRL Protocol Analyzer we were able to show that this formatting was advisable in order to prevent a denial-of-service attack.

We discovered another anomaly when analyzing the virginity requirement for key agreement keys. In this case, we discovered a path that resulted in a key agreement key from the opposite party to be accepted twice. This did not lead to an attack on the security of the protocol, assuming that the accepting party generates fresh key agreement keys on her own side, but it did show how the protocol could fail under certain circumstances. It also pointed out a weakness in our requirements under certain conditions.

The situation is as follows. We assume that A first initiates a conversation with B, and then with I.

1. *A sends to B* : $Cert_A, N_A, alglist_A$
2. *B sends to A* : $Cert_B, \{AK_B\}_{K_A}, algchoice_B, \{md(\{AK_B\}_{K_A}, algchoice_B, N_A, alglist_A)\}_{K_B^{-1}}$
3. *A sends to I* : $Cert_A, N'_A, alglist_A$
4. *I sends to A* : $Cert_I, \{AK_B\}_{K_A}, algchoice_I, \{md(\{AK_B\}_{K_A}, algchoice_I, N'_A, alglist_A)\}_{K_I^{-1}}$
5. *A sends to I*: $\{AK'_A\}_{K_I}, \{md(\{AK'_A\}_{K_I}, \{AK_B\}_{K_A})\}_{K_A^{-1}}$
6. *A sends to B*: $\{AK_A\}_{K_B}, \{md(\{AK_A\}_{K_B}, \{AK_B\}_{K_A})\}_{K_A^{-1}}$

In Step 6, A accepts AK_B as a good key agreement key from B, even though she had already accepted it as a good key agreement key from I in Step 5.

Note that it would appear that such a protocol would fail BAN analysis. As a matter of fact, it does not, and BAN analysis is successfully applied to the protocol in [3]. This is because BAN's definition of freshness is somewhat weaker than our virginity requirement. Freshness requires only that the key was not used before the current run of the protocol. This is certainly true in this case, since the conversation between A and I begins after the conversation between A and B.

Although A accepts AK_B twice, this does not appear to cause any problems in the Aziz-Diffie protocol. The intruder does not learn AK_B, since it is encrypted with A's public key. Moreover, A takes the exclusive-or of AK_B with a different key agreement key generated by herself each time she uses it. However, it is possible to imagine scenarios where such a situation causes problems. Suppose, for example, that the requirements of the protocol were relaxed so that A was allowed to use the same key agreement key generated by herself more than once, perhaps because her computational resources were limited. Then the session keys generated by A to converse with I and B would be the same. Suppose also the crypto-algorithm was one, such as a one-time pad or a stream cipher, in which use of the same key more than once could result in compromise of the key. Then, if A was tricked into using the key to encrypt messages to I and messages to B, I might be able to use the information to recover the key, and hence the traffic between A and B.

This combination of assumptions appears to be rather unlikely. Nevertheless, it is still useful to know under what circumstances it is safe to use the protocol, even if we have not uncovered a serious weakness.

Another interesting result we found was, that, if we want to prevent anomalies such as the one we described above, our virginity requirement is not strong enough. Indeed, if we reverse steps 5 and 6 in the above attack, it now satisfies the virginity requirement, since when A accepts the key as coming from honest B, it has not previously been accepted by anyone else. A better requirement would be to say, that, if A accepts as a key good for communication with an honest party B, it should not have previously been accepted by any honest party for communication with any one except possibly B for communication with A, and, if A accepts a key as good for communication with a dishonest party, it should not previously been accepted by any honest party for communication with another honest party.

For key agreement protocols, this requirement for session keys would be expressed as follows for the initiator of the protocol:

$$\begin{aligned}&\texttt{init\_accept\_sk}(user(A, honest), user(B, honest), K, M1) \rightarrow\\ &\quad \neg(\diamondsuit\texttt{init\_accept\_sk}(user(C, honest), user(D, X), K, M?)\wedge\\ &\quad (\diamondsuit\texttt{rec\_accept\_sk}(user(C, honest), user(D, X), K, M?) \rightarrow (C = B \wedge D = A))\end{aligned}$$

$$\begin{aligned}&\texttt{init\_accept\_sk}(user(A, honest), user(B, dishonest), K, M1) \rightarrow\\ &\quad \neg(\diamondsuit\texttt{init\_accept\_sk}(user(C, honest), user(D, honest), K, M?)\wedge\\ &\quad \neg(\diamondsuit\texttt{rec\_accept\_sk}(user(C, honest), user(D, honest), K, M?)\end{aligned}$$

Similar requirements can be developed for the receiver's accepting a session key, and for the sender's and receiver's accepting of a key agreement key from the other party.

5 Conclusion

In this paper we have set forth a formal language for specifying requirements of cryptographic protocols in terms of conditions on sequences of events. These requirements can then be mapped to a protocol specification in the NRL Protocol Analzyer language, after which the NRL Protocol Analyzer can be used to determine whether or not the protocol satisfies the requirements. We developed a set of generic requirements on key agreement protocols, and applied them to a sample protocol, the Aziz-Diffie key agreement protocol for mobile communication. Our use of the NRL Protocol Analyzer to analyze the protocol with respect to the requirements has shown that, not only can it be useful for determining whether or not requirements can be met, but also for uncovering the assumptions that are necessary to determine whether or not the requirements have been met, and for discovering limitations of the requirements themselves.

Acknowledgements

We are grateful to Ashar Aziz for useful discussions on the Aziz-Diffie protocol and to Paul Van Oorschot for pointing out the ElGamal protocol to us.

References

[1] Martín Abadi. An Axiomatization of Lamport's Temporal Logic of Action. Research Report 65, Digital Systems Research Center, October 1990.

[2] Martín Abadi and Mark Tuttle. A Semantics for a Logic of Authentication. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 201–216. ACM Press, August 1991.

[3] A. Aziz and W. Diffie. Privacy and Authentication for Wireless Local Area Networks. *IEEE Personal Communications*, 1(1):25–31, 1994.

[4] J. Burns and C.J. Mitchell. A Security Scheme for Resource Sharing Over a Network. *Computers and Security*, 9:67–76, February 1990.

[5] Michael Burrows, Martín Abadi, and Roger Needham. A Logic of Authentication. Research Report 39, Digital Systems Research Center, February 1989. Parts and versions of this material have been presented in many places including *ACM Transactions on Computer Systems*, 8(1): 18–36, Feb. 1990. All references herein are to the SRC Research Report 39 as revised Feb. 22, 1990.

[6] Michael Burrows, Martín Abadi, and Roger Needham. Rejoinder to Nessett. *Operating Systems Review*, 24(2):39–40, April 1990.

[7] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[8] D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, March 1983.

[9] Taher ElGamal. A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[10] Robert Goldblatt. *Logics of Time and Computation, 2^{nd} edition*, volume 7 of *CSLI Lecture Notes*. CSLI Publications, Stanford, 1992.

[11] Leslie Lamport. A Temporal Logic of Action. Research Report 57, Digital Systems Research Center, April 1990.

[12] C. Meadows. A System for the Specification and Analysis of Key Management Protocols. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 182–195. IEEE Computer Society Press, Los Alamitos, California, 1991.

[13] C. Meadows. Applying Formal Methods to the Analysis of a Key Management Protocol. *Journal of Computer Security*, 1:5–53, 1992.

[14] S.P. Miller, C. Neuman, J.I. Schiller, and J.H. Saltzer. Kerberos Authentication and Authorization System. In *Project Athena Technical Plan*. MIT, Cambridge, Mass., July 1987.

[15] D. M. Nessett. A Critique of the Burrows, Abadi, and Needham Logic. *Operating Systems Review*, 24(2):35–38, April 1990.

[16] S.G. Stubblebine and V.D. Gligor. On Message Integrity in Cryptographic Protocols. In *Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 85–104. IEEE Computer Society Press, Los Alamitos, California, 1992.

[17] Paul Syverson and Catherine Meadows. A Logical Language for Specifying Cryptographic Protocol Requirements. In *Proceedings of the 1993 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 165–177. IEEE Computer Society Press, Los Alamitos, California, 1993.

[18] Paul Syverson and Catherine Meadows. Formal Requirements for Key Distribution Protocols. In *Pre-proceedings of EUROCRYPT '94*, pages 325–337, University of Perugia, Italy, May 1994. Final proceedings forthcoming from Springer-Verlag.

[19] Paul F. Syverson. The Use of Logic in the Analysis of Cryptographic Protocols. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 156–170. IEEE Computer Society Press, Los Alamitos, California, 1991.

[20] Paul F. Syverson. On Key Distribution Protocols for Repeated Authentication. *Operating Systems Review*, 27(4):24–30, October 1993.

[21] Johan van Bentham. *The Logic of Time*, volume 156 of *Synthese Library*. Kluwer Academic Publishers, Dordrecht, The Netherlands, second edition, 1991.